ProtocolV2

The TextSecure encrypted messaging protocol is an end-to-end encrypted messaging protocol with deniability guarantees and message-level forward secrecy, similar to OTR Messaging. Version 2 of the TextSecure messaging protocol uses the **no header keys variation** of the axolotl ratchet and protobuf records. This is a departure from Version 1 of the TextSecure messaging protocol, which used the OTR ratchet and custom binary structures.

Message Formats

WhisperMessage

```
# Protocol Buffer
message WhisperMessage {
  optional bytes  ephemeralKey    = 1;
  optional uint32 counter         = 2;
  optional uint32 previousCounter = 3;
  optional bytes  ciphertext      = 4;
}

struct {
  opaque version[1];
  opaque WhisperMessage[...];
  opaque mac[8];
} TextSecure_WhisperMessage;
```

The `WhisperMessage` protobuf consists of the following fields:

1. "ephemeralKey". This is an ephemeral Curve25519 key for the message's current DH ratchet. This corresponds to `DHR` in the axolotl protocol description.
2. "counter". This is a monotonically incrementing counter for each message transmitted under the same "ephemeralKey". This corresponds to `N` in the axolotl protocol description.
3. "previousCounter". This is the max value of the counter that was transmitted under the sender's last "ephemeralKey." This corresponds to `PN` in the axolotl protocol description.
4. "ciphertext". This is the ciphertext body of the message, encrypted with a message key derived according to axolotl using a 256bit AES cipher in CTR mode with the high 4 bytes of the counter corresponding to the "counter" value transmitted with this message.

The `TextSecure_WhisperMessage` structure is what is transmitted by a client, and it consists of:

1. "version". A one byte version identifier, with the high 4 bits representing the current version of the message and the low 4 bits representing the maximum protocol version the client knows how to speak.
2. "WhisperMessage". A serialized `WhisperMessage` protocol buffer (above).
3. "mac". An HMAC-SHA256 of both `version` and `WhisperMessage` concatenated, then truncated to 8 bytes.

PreKeyWhisperMessage

```
# ProtocolBuffer
message PreKeyWhisperMessage {
```

```
  optional uint32 preKeyId    = 1;
  optional bytes  baseKey     = 2;
  optional bytes  identityKey = 3;
  optional bytes  message     = 4;
}

struct {
  opaque version[1];
  opaque PreKeyWhisperMessage[...];
} TextSecure_PreKeyWhisperMessage;
```

The PreKeyWhisperMessage protobuf consists of the following fields:

1. "preKeyId". The ID of the client's prekey that was retrieved by the sender.
2. "baseKey". The base Curve25519 key exchange ephemeral. This corresponds to `A0` in the axolotl protocol description.
3. "identityKey". The Curve25519 identity key of the sender. This corresponds to `A` in the axolotl protocol description.
4. "message". This corresponds to a full serialized `TextSecure_WhisperMessage` that contains the actual encrypted message.

The `TextSecure_PreKeyMessage` is what is transmitted by a client, and consists of the following fields:

1. "version". A one byte version identifier, with the high 4 bits representing the current version of the message and the low 4 bits representing the maximum protocol version the client knows how to speak.
2. "PreKeyWhisperMessage". A serialized `PreKeyWhisperMessage` protocol buffer (above).

KeyExchangeMessage (SMS Transport Only)

```
# Protocol buffer
message KeyExchangeMessage {
  optional uint32 id           = 1;
  optional bytes  baseKey      = 2;
  optional bytes  ephemeralKey = 3;
  optional bytes  identityKey  = 4;
}

struct {
  opaque version[1];
  opaque KeyExchangeMessage[...];
} TextSecure_KeyExchangeMessage;
```

The KeyExchangeMessage protocol buffer consists of the following fields:

1. "id". This is a multipart field. The bottom 5 bits correspond to a "flags" bitmask, and the bits above the first five represent a sequence number.
2. "baseKey". This is a Curve25519 ephemeral key corresponding to `A0` in the axolotl protocol description.
3. "ephemeralKey". This is a Curve25519 ephemeral key corresponding to `A1` in the axolotl protocol description.
4. "identityKey". This is a Curve25519 identity key corresponding to `A` in the axolotl protocol description.

The `flags` bitmask in the `id` field contains the following masks:

- `INITIATE_FLAG = 0x01`. This flag indicates that the key exchange message is an "initiate." The sequence number for this `id` should be randomly generated to be >=0 and <=65535.
- `RESPONSE_FLAG = 0X02`. This flag indicates that the key exchange message is a response to an "initiate." The sequence number for this `id` should be set to the value of the sequence number in the received initiate `id`.
- `SIMULTAENOUS_INITIATE_FLAG = 0x04`. This flag indicates that the sender is responding to an initiate, but already had an unresponded initiate outstanding, and that this might be a "simultaneous open" situation. The `RESPONSE_FLAG` must always be set when this flag is set.

The `TextSecure_KeyExchangeMessage` is what is transmitted by a client, and consists of the following fields: 1. "version". A one byte version identifier, with the high 4 bits representing the current version of the message and the low 4 bits representing the maximum protocol version the client knows how to speak. 1. "KeyExchangeMessage". A serialized `KeyExchangeMessage` protocol buffer (above).

Axolotl Implementation

The TextSecure V2 protocol implements the Axolotl ratchet. This section specifies the implementation-specific details.

Defining Roles

The Axolotl session initialization requires determining who is Alice and who is Bob. TextSecure has two cases:

1. In the PreKeyWhisperMessage case, where a client retrieves a PreKey for another client and sends a PreKeyWhisperMessage all at once, the sender of the message is Alice and the receiver of the message is Bob.
2. In the KeyExchangeMessage case, both parties compare the base keys (`A0` and `B0`) they have exchanged. The party with the smaller base key is Alice, the party with the larger base key is Bob.

Initial Root Key

The initial root key is derived from 3DHE of the tuple of the client base keys and identity keys.

The 3DHE secrets are concatenated before being fed into a KDF. The concatenation order is:

```
if (alice) {
  ECDHE(theirEphemeral, ourIdentity)
  ECDHE(theirIdentity, ourEphemeral)
  ECDHE(theirEphemeral, ourEphemeral)
} else {
  ECDHE(theirIdentity, ourEphemeral)
  ECDHE(theirEphemeral, ourIdentity)
  ECDHE(theirEphemeral, ourEphemeral)
}
```

The concatenated ECDHE shared secrets are then fed into HKDF to derive a 32 byte root key (`RK`) and 32 byte chain key (`CK`). HKDF is used with a `salt` of zero bytes and an `info` of the octet string "WhisperText".

The first 32 bytes out of the HDKF are used for the root key (`RK`) and the second 32 bytes out are used for the chain key (`CK`).

Chain Key Derivation

A chain key (`CK`) is used both to derive the next chain key as well as to derive the message key (`MK`) that corresponds to that chain key.

To derive the message key:

```
message_key = HMAC-SHA256(chain_key, 0x01)
```

To derive the next chain key:

```
next_chain_key = HMAC-SHA256(chain_key, 0x02)
```

Message Key Derivation

A message key (`MK`) is fed into HKDF to derive a 32 byte cipher key and a 32 byte mac key. The HDKF "input key material" is the message key (`MK`), the salt is all 0x0 bytes, and the `info` is the octet string "WhisperMessageKeys".

Root Key Derivation

Each new remote ephemeral key received (`DHR`) triggers the generation of a new root key (`RK`) and new sender chain key (`CKs`).

HKDF is used to produce a new root key and chain key. The HKDF `input key material` is the ECDHE(theirEphemeral, ourEphemeral) shared secret, the HKDF `salt` is the current root key (`RK`), and the HKDF `info` is the octet string "WhisperRatchet".

The first 32 bytes out of the HKDF are the new root key (`RK`) and the second 32 bytes out of the HKDF are the new sender chain key (`CKs`).

SMS Transport Format

When the data transport is not available and messages are transmitted via SMS, it is not possible to rely on the multipart message facility provided by the SMS UDH in order to send long messages. It does not work on CDMA networks, so we have to provide for message chaining ourselves. Messages are fragmented using the following format:

```
struct {
  opaque identifier[3];
  TextSecure_Version version;
  opaque fragment_count[1];
  opaque multipart_message_id[1]; (optional)
  TextSecure_Message message;
} TextSecure_Fragment;
```

- "identifier" Everything following this field is appended to an identifier (the string "?TSK" for a key exchange message or the string "?TSM" for a data message) and hashed iteratively with SHA1 for 1000 iterations. The result is then truncated to 3 bytes.
- "fragment_count" The top 4 bits represent the index of this fragment (0-based) and the bottom 4 bits represent the total number of fragments.
- "multipart_message_id" This is a unique identifier to group message fragments together. It is only present if there is more than one message fragment in total.
- "message" The `TextSecure_WhisperMessage` , fragmented at the SMS size boundary. In order to avoid wasting space in the first fragment, when the complete encrypted and MAC'd application-level message is handed down to the transport layer, the version number is stripped out of the `TextSecure_WhisperMessage` and put on the front of each transport-level fragment piece. Once the fragments are re-assembled, the version number is put back on the front of the `TextSecure_WhisperMessage` and passed along to the application-layer as a fully reconstructed message.

The entire `TextSecure_Fragment` is Base64 encoded, without trailing padding.

Sending A Message

When composing a message, the total headerfied, encrypted, and mac'd data message is handed to the transport layer.

If the transport layer is SMS, it then splits the data message into fragments small enough to fit into individual SMS messages. Once the receiving end has received all the fragments, it reassembles them into one message again before verifying the MAC and decrypting.

Some knowledge of the transport layer sizing requirements is needed by the application layer, since it is responsible for padding the message out. What is handed to the transport layer should be sized such that when split into fragments and fully base64 encoded, it will exactly fill the maximum available payload space in each SMS message (140 bits, 160 characters).

In the end, there are 60 characters available in the first fragment, and 115 in each subsequent fragment.

Local Encryption

Should clients choose to have "non-ephemeral" conversations, which is to say that the messages they send and receive are stored to disk, they need to be re-encrypted since the original keys used to encrypt them may vanish as the protocol rolls forward.

Clients generate a 128bit AES key and 160bit MAC key at install time, which are hashed and written to disk using PBKDF2.

Locally encrypted messages are then encrypted using AES-CBC$ with HMAC-SHA1 in the encrypt-then-authenticate paradigm. So the format is:

```
struct {
  opaque random_iv[16];
  opaque encrypted_message[...];
  opaque mac[20];
} TextSecure_LocalMessage;
```

- "random_iv" is a random IV.
- "encrypted_message" is the ciphertext of AES-CBC$(plaintext)
- "mac" is HMAC-SHA1 of the ciphertext and IV.

Last edited by Moxie Marlinspike, 14 days ago